exec magic

Purpose: explain how best to start an executable script under Unix. From news:comp.lang.tcl See also: DOS BAT magic

But before you invest some of your precious time checking out what's in store at DOS BAT magic, be informed that you can use Cygwin's port of sh instead, and avoid some MS-DOS Hell.

There seems to be a growing consensus that the best way to make a Tcl script executable as if it were a Bash or Sh script is to put the following on the very top line of your Tcl (or Tk) script:

```
#! /usr/bin/env tclsh
```

If you're using Tk, then you may want to invoke "wish" instead of "tclsh", like so:

```
#! /usr/bin/env wish
```

though even better is

```
#! /usr/bin/env wish
package require Tk
```

In this way, not only would the script work with wish, but also tclsh's and tclkit's with GUI support. (Note however that loading Tk into a vanilla tclsh will not work under all circumstances. For example, there is a "console only" version of tclkit that doesn't support Tk.)

Note that the #! line must appear at the top of your Tcl program file, and the file must have execute permissions.

The Tcl/Tk file can now be invoked just as if it were a command file, or a shell script written in Bash, sh, csh, et cetera. After seeing the first line, the shell should pass the rest of the script to tclsh/wish.

The normal function of /usr/bin/env is to set one or more environment variables before calling some other program, but in this case we just call the other program (tclsh or wish) without setting any. The point of the whole thing is instead that env when calling the other program does so using a system call that looks for the named program in the directories of the PATH environment variable, just like a shell would. Therefore, the first "tclsh" (or "wish") command found in your current PATH will be the tclsh version used to run your Tcl script, and the program will have the exact same environment settings it would have if you'd invoked the script by typing "tclsh myfile.tcl" from your normal shell prompt.

There is nothing special about the env program in this case, other programs that should work just as well are:

```
/usr/bin/command
/usr/bin/nice
```

with **command** perhaps being the canonical (but with a slightly longer name) program for locating and running another program. The functionality can also be had from

```
/usr/bin/nohup
/usr/bin/time
```

but these do not seem to have a neutral mode of operation (so using these would always have side-effects).

Unsigned, but judging from IP number probably willdye: The "env" technique is a little surprising only because for a long time, the accepted way to invoke an executable Tcl script under Unix was to invoke "#!/bin/sh" with some special continuation lines and commands, as shown below. Further on down this wiki entry, you should also find a discussion of the "env" technique, and why it's better.

If someone ever finds a problem with the "env" technique, please post it here. Otherwise, we should probably update all of the Tcl/Tk documention to deprecate the older (rather convoluted) continuation-line trick.

Ah, but what if you have multiple versions of Tcl/Tk installed, and you want to invoke a particular version? You can do it by changing your PATH environment variable so that the preferred version is found first. That will affect every script which uses the same PATH value. If you know that a particular script needs a particular version of tclsh/wish, you could update the script so that the first line points to the version you want to execute instead of the first tclsh/wish found in PATH. For example, if you named the 8.5 version of wish as "wish8.5", you could change the first line to "#!/usr/bin/env wish8.5".

rdt See my comment about the use of /usr/bin/env below.

In some cases, the continuation-line trick still has some advantages over /usr/bin/env. Consider this clever-albeit-somewhat-obtuse scheme by FPX:

FPX The spiel of having a shell script at the beginning of your Tcl script can of course be extended as necessary. This is what I use to make my scripts run both on Unix and from within Cygwin's bash shell on Windows.

```
        #! /bin/sh
        # the next line restarts using wish8.4 on unix \
        if type wish8.4 > /dev/null 2>&1 ; then exec wish8.4 "$0" ${1+"$@"} ; fi
        # the next line restarts using wish84 on Windows using Cygwin \
        if type wish84 > /dev/null 2>&1 ; then exec wish84 "`cygpath --windows
$0`" ${1+"$@"} ; fi
        # the next line complains about a missing wish \
        echo "This software requires Tcl/Tk 8.4 to run." ; \
        echo "Make sure that \"wish8.4\" or \"wish84\" is in your \$PATH" ; \
        exit 1
```

This scheme could be extended to search the $PATH for any usable version of tclsh or wish.

Here's the original text about the continuation line trick, before /usr/bin/env became more popular:

```
#!/bin/sh
# This line continues for Tcl, but is a single line for 'sh' \
exec wish "$0" ${1+"$@"}
```

(beginning of original message)

```
Subject: Re: Documentation error (was: Why "$@" and not $@ ?)
From: [email protected]
Date: 1998/07/07
Newsgroups: comp.lang.tcl
John> But back to the topic ... Maybe it would help if we could get a
John> good writeup of just how and why it works, and twist arms to
John> install the description in various highly-visible places.  I'd
John> do it myself, but I'm not at all confident that I could write a
John> correct description of it all. Maybe I should give it a try, and
John> post it for criticism ...
```

The incantation looks like this:

```
        exec wish "$0" ${1+"$@"}
```

Here's how it works:

Each argument to sh is numbered $1, $2, $3, ... $9.

$0 is the name of the script. We put it in quotes to preserve whitespace and other weird characters.

The special variable $* expands to all the arguments. However, it expands *unquoted*, meaning that spaces will be lost in the expansion.

This means that the naive:

```
        exec wish "$0" $*
```

will not work properly if any argument has a space in it. Likewise, using "$*" will fail because that will expand to a single argument.

The special form "$@" was introduced (the quotes are part of it) to work around this problem. This form expands to all the arguments, but properly (and individually) quoted. So why doesn't the following attempt work?

```
exec wish "$0" "$@"
```

The reason is that some versions of sh have a bug. If there are no arguments, these broken sh's will still expand "$@" into a single empty argument. Oops.

The sh syntax ${VAR+VALUE} expands to VALUE if $VAR is set, and nothing otherwise. We abuse this syntax to work around the buggy sh implementations. Recall that the first argument is named $1. If it exists, then we use "$@"; otherwise we do nothing:

```
exec wish "$0" ${1+"$@"}
```

The above is my interpretation. No doubt further refinements and clarifications are in order.

Tom

DKF - On some platforms, it is important to put a space between the *#!* and the */bin/sh* (or whatever) so that magic file numbering works. More sophisticated versions of this trick are needed when you've got a plethora of Tcl versions installed and your script cares which one it gets. These extensions are pretty much obvious to a shell programmer though...

JBR - The above comment intrigued me since #! is actually an executable magic number and there are no 32 bit magic values. Wikipedia[L1] reveals that this appears to be a bug in the autoconf docs copied from unimplimented (but documented) features planned for 4.2BSD[L2].

LV - note that the above is shell specific ; if you use something other than Bourne/korn shell, or older versions of these two, your experiences may differ in what works. Certainly if you move off into more **futuristic** shells, you may find that none of these features work.

RS - But I think there is a habitual agreement that /bin/sh can always be expected, so it's safest to base portable scripts on that.

LV - But at least on Linux these days, it is my understanding that /bin/sh is really either ksh or bash... if one finds the above magic doesn't work, just drop by here and let us know what is going on and perhaps someone can help find new magic for that platform.

DKF - All unix shells come from two lineages: Bourne Shell and C-Shell. The two are almost completely incompatible with each other at a syntactic level (though most modern shells derive from both, at least feature-wise) so when you ask for /bin/sh, you will always

get something that knows how to handle Bourne syntax (which is all the above trick depends on). Correspondingly, when you ask for /bin/csh, you always get something that knows C-Shell syntax. The put-a-space-after-the-#! trick is a deeply obscure feature of Unix kernel magic processing; you only get to find these things out by reading background material for autoconf...

willdye According to a Bash expert, the trick should work for any version of any Bourne shell. If not, there's a bug in the shell.

More commentary appears in [L3].

Donal Fellows contributed in c.l.t. an extended version of the magic code, that can call either *tclsh* or *wish*:

```
#! /bin/sh
# If the script's name starts with tk, use wish... \
case "`basename $0`" in *tk*) exec wish "$0" ${1+"$@"}
# ... and otherwise use tclsh. \
;; *) exec tclsh "$0" ${1+"$@"};;
# That is all. \
esac
```

DKF notes in passing: Did I? I don't remember that; it's really quite clever. I'm impressed with myself...

Wish has a few options like -s that it snatches from the command line before setting up *argv*. If you want those to be visible for your script (and not for wish itself), insert doubledash (which ends wish's argument processing):

```
        exec wish "$0" -- ${1+"$@"}
```

Note that if you do that, on Unix systems your script will no longer handle X windows options (like -geometry and -display) automatically.

RS found that the first line

```
#!/usr/bin/env tclsh
```

also works (starts the first tclsh in PATH) quite nicely on his Linux, Solaris, and Cygwin installations - so he will use this instead of *exec magic* in the future.

Note that this actually does the same as using */bin/sh*. That trick (abusing the different comment handling of *sh* and *tclsh* **is** a trick) with */bin/sh* does nothing else than setting up the environment (including PATH) so that *tclsh* is a valid command that actually runs something. If you look at *man env* you will see that *env* does nothing else than setting up

the environment and execute the command given to it, along with remaining arguments. Using *#!/usr/bin/env tclsh* is just the **correct way** to achieve this (simple) goal. It is meant for exactly the task in question. - joh

rdt 2006.07.15 - I'd say, joh, thats a matter of opinion. In my more that 2 decades of Unixy/Linux experience, I've seen env in at least four different situations:

1. not present on the machine,
2. only in /usr/bin,
3. only in /bin,
4. in both /bin and /usr/bin (usally as a link from one to the other).

Now, the **one** common thing (but I'm sure there are exceptions), was that they **all** had a /bin/sh. So the probability of of the /usr/bin/env working is less than that of /bin/sh magic. If you, or others, have always found /usr/bin/env, then *lucky you*.

SEH 20060715 -- My experience has been the opposite. In work situations where I had to run scripts on a wide variety of Unices and versions, there were computers with no /bin/sh, but I never encountered one where env was not in /usr/bin. So discovering the env trick was a lifesaver for me. I guess it's best to know about both options and use what works out best for you.

Lars H, 2008-06-20: Under the Filesystem Hierarchy Standard, /bin/sh is mandatory. The env program is not even mentioned. Interestingly enough, it also lists /usr/bin as the proper location for tclsh, wish, and expect (if they are at all installed).

Some people run into problems if the file was created under DOS first and then used on Linux. The problem is this. Say you have a file that looks like this:

```
#! /bin/sh
# This line continues for Tcl, but is a single line for 'sh' \
exec wish "$0" ${1+"$@"}
```

or some variation. However, originating in DOS/Windows, the lines are all terminated by carriage return (^M character). When the user executes the file, s/he might get the error: ksh: /tmp/err.tcl: not found

The trailing carriage return causes the command shell problems before it ever reaches tcl. The user will however think there's a tcl problem - so this is one of the things to check.

RPR (2006AUG27) Actually, in DOS/Windows, the lines in a pure text file terminate in carriage-return linefeed (^M^J, or hex 0D 0A).

I hope it is ok to edit this. I am a beginner at tcl and programming (half an hour's playing with it). However I found that I could get a gui tcl script (the example that was given for buttons) to run from a double click in konqueror on Mandrake 9.2 by right clicking on the

file and chosing "open with" and then chosing "wish". I wondered if this would be useful information for other newbies?

FW: Good to know, I suppose. Yes, you can edit anything you want - that's the point :)

FW: So let's be clear - does the *#!/usr/bin/env tclsh* method have any disadvantage whatever over exec magic?

MSW(2005-05-03): *yup*. The "magic" is more portable. You can't count on env being installed. The "magic" has seen years of testing and wide deployment, you can trust it. You can't say the same about env (although supposedly the last dinosaurs which lack it are dying).

rdt Yes, I prefer the *magic*. See my comment about this use above.

D. McC: Almost as soon as I read this three weeks ago, I scrapped the "exec magic" at the beginning of the Tcl/Tk programs I've written and inserted *#!/usr/bin/env wish* at the beginning of my working versions instead. They work fine. SnackAmp, WaveSurfer, and FileRunner also have no problem with this line substituted for "exec magic" on my Mandrake Linux 9.2 system. The only disadvantage I see is that Konqueror no longer thinks these programs are shell scripts, so I can't run them with a double-click from Konqueror any more. I can easily live without that. So I say, good-bye forever to evil-looking exec magic!

LV At the very least, if you are going to ignore the fact that env might not exist in /usr/bin, consider this change to what D. McC is doing - use

```
#!/usr/bin/env tclsh
package require Tk
```

if you are using a Tcl from the past 5 years or so. That way, even if the person using the program is using a tclkit instead of tclsh, the program still works.

RJ 2005-05-03 - Here's something that may not belong on this page, but it's as close as I can get. Consider an application that will run on Solaris 8, Tcl v8.3+, Expect v5.31+ and Tk 8.3+. The application will also require the capability for the user to enter a "-d" on the command line as a *nix-like switch. For example: *dothis -d tothat*. The obvious solution would be to use -

```
#!/usr/bin/env expectk --
```

as the magic. This would load Tcl, Tk and Expect into the interpreter and pass whatever else is on the command line on to $argv, right? Right. It does, BUT it also passes the "-d" on to expectk as an argument. This results in:

```
Application initialization failed: ambiguous option "-d"
```

..because expectk wants to see either -display or -diag. The expectk (at least as it's compiled here) ignores the "--" switch termination. [Note that this occurs no matter which form of exec magic I use.]

I've tried everything to make this work with expectk and so far (this is a month now) I have only found one thing to make this work...

I load Expect and Tcl with -

```
#!/usr/bin/env expect --
```

..which behaves correctly and ignores switches after the "--" passing them on to the application as $argv.

But this still leaves me without Tk. As soon as I package require it, it dies with:

```
Application initialization failed: "-d" option requires an additional argument
```

..I assume Tk is considering it the -display option.

The workaround then became to parse the command line args to variables in the app first, set argv to null, then call Tk. Which even to my minimal scripting standards is an embarrassing hack.

Is this a seriously flawed compilation or bug?

JAB 2005-09-20 - Here's something broken with exec magic replacement: the above

```
#!/usr/bin/env expect --
```

doesn't work for me. (RHEL3, OS X) Env complains: "/usr/bin/env: expect --: No such file or directory" Obviously env isn't treating the -- as an argument to expect, so it's not doing what it should according to the man page or the above post. What system is it working correctly on for you?

The standard magic doesn't do what I want to either - preserve user input to the script of -- such as "./test -- -foo". I modified it to

```
#!/bin/sh
#\
exec expect "$0" -- ${1+"$@"}
```

And that does preserve my "--" argument to the script.

AMG: The reason is that the #! line can only pass one argument to the interpreter (/usr/bin/env in this case). Attempting to supply multiple arguments will give you one long argument with spaces in the middle of it. As the error message shows, /usr/bin/env's first argument is "expect --", and there's no program with that name. Not like it matters, but its second argument is the name of the script file being executed.

The multiline comment polyglot /bin/sh trick seems to be the best way to stuff more than one argument into the interpeter. This is needed is when passing the -encoding utf-8 arguments to tclsh.

AMG: I prefer to make the continued comment line contain an explanation that the **next** line restarts with tclsh. This hopefully discourages people from separating the two.

```
#!/bin/sh
# The next line restarts with tclsh.\
exec tclsh "$0" ${1+"$@"}
```

Also, here [L4] is some exec magic to start with the "latest version of tclsh in PATH". Personally I don't see the need for this, but it is a complex example of exec magic.

One note about many/most of these techniques. They require the script writer to know what the name of the stand alone tcl interpreter is on someone else's system.

Why is this important?

Because there are always going to be exceptions. So none of these are *fool proof*.

For instance, ActiveTcl installs the tclsh and wish on SPARC Solaris as tclsh$version.$level (and wish$version.$level) where version is the major version number and level is the minor version level. Some sites may only have a tclkit available - which could be instaleld as tclkit, tclkit.exe, tclshkit.exe or something with a version number in it.

Also, all the cases above in which tclsh/wish/expect and so forth are invoked as simple names, rather than pathnames have the side effect of getting whatever version happens to be encountered first in the PATH - and that very well will vary from user to user even on a particular system! This will make debugging problems more difficult. However, using a full pathname in the first line doesn't guarantee that is the tclsh that will be used with the script - the user could have invoked the program as:

```
/path/to/weird/tclsh yourscript.tcl
```

The starpack technology is a way to take a tcl script, merge it with a tcl interpreter, and create a stand-alone file that doesn't care where or if tcl is installed. The down side is that each of these starpacks are installed with the tcl interpreter embedded.

There is an old addage that I read many years ago - **TANSTAAFL** (There ain't no such thing as a free lunch).

LV During the last week of Sept, 2008, on comp.lang.tcl, this question came up:

```
I've used Tcl/Tk for year under Linux.    At the top of the script I
always used something like:

#!/usr/bin/wish

gave the script execution permissions and could run things by directly
typing the script name.

However this doesn't appear to work on my Macbook.  So far it
behaves like it is just ignoring the #! line and trying to interpret
the Tcl/Tk commands as sh commands.  If I run my script as:

wish fun.tcl

all works.
```

After a bit of discussion, this remark came up

Andreas Leitgeb writes:

That topic came up here not long ago. The reason why it didn't work was, that "wish" was itself a shell-script, and while linux has no problem with scripts being interpreters for other scripts, most unices (including mac) do not support that. And then, for extra bonus, wish isn't even a shellscript on linux. Don't know, why it is one on mac. (they wrote it was) Using the binary tclsh (with whatever appropriate full path) as interpreter and loading tk dynamically: package require Tk seems to avoid the problem.

Lars H: That just gives me "can't find package Tk"; Tk isn't (yet) a loadable package on OS X. That there are two possible windowingsystems (X11 and aqua) may be part of why…

In case anyone is curious, the /usr/bin/wish8.5 script just looks as follows:

```
#!/bin/sh
"$(dirname
$0)/../../Library/Frameworks/Tk.framework/Versions/8.5/Resources/Wish.app/Contents/
 "$@"
```

Daniel Steffen continued:

it needs to be a script that execs the real aqua Wish binary for the interaction with the windowserver to work correctly (don't ask...)

use the standard exec magic

```
#!/bin/sh
# \
exec wish "$0" "$@"
```

which has been recommended forever to avoid (among other reasons) exactly this issue... (tclsh or wish could be a script on other platforms as well, e.g. in a shared environment where an executable has to be chosen based on architecture)

the slightly less portable (and less magical)

```
#!/usr/bin/env wish
```

also works to avoid the issue

Updated 2015-03-01 20:58:16